P R O J E C T

Martina Baga (16055A)
Sara Cinquini (27283A)

Creation of a NoSQL DataBase for Credit Card Fraud Detection

A . A 2022-2023

Summary

Introduction

Context

Payment card fraud is a major challenge for business owners, payment card issuers and transactional services companies because every year these frauds cause substantial and growing financial losses. Many Machine Learning approaches have been proposed in the literature for making more and more automatic the process of identifying fraudulent patterns from large volumes of data. The book "Machine Learning for Credit Card Fraud detection – Practical handbook"[1] reports different approaches for facing the issue and for evaluating the quality of the proposed prediction results.

For the purpose of this project we are not interested in the application and verification of the ML approaches, however we have exploited the transaction data simulator code (reported in Section 2 of Chapter 3 of the cited book) for the generation of data stored in a NoSQL database.

Other informations

We have uploaded the folder related to our project in the following github repository:
https://github.com/cinqusara/fraud_detection_dbmss.git

[1] https://fraud-detection-handbook.github.io/fraud-detection-handbook

Python code

Applied changes

These datasets have been generated by the common script provided by the cited book and adapted to our goal. In particular, we have decided to transform the **.pkl** files into the **.csv** format, making them easier to use. The following lines of code generate the .csv files for each table created (*customer_profiles_table*, *terminal_profiles_table*, *transactions_df*):

```
current_dir = os.getcwd()
current_dir
```

```
path = current_dir + '\\csv\medium'
os.chdir(path)
os.getcwd()
```

```
customer_profiles_table.to_csv('customer_profiles_table.csv', index=False)
terminal_profiles_table.to_csv('terminal_profiles_table.csv', index=False)
transactions_df.to_csv('transactions_df.csv', index=False)
```

For all the datasets we have decided to use a specific starting date (*2022-10-03)*, which refers to the starting date of the New generation data models and DBMSs' course. Furthermore, for generating the transactions, we have to set a number of days, in our project 365.

Consideration

Following the example reported by the generator code, the number of terminals always exceeds the number of customers. Although in a real scenario it would be more plausible to use a larger number of people.

Used tools

For this project we have used a computer with these characteristics:

- 16 RAM
- intel CORE i7

This information could affect the analysis of the execution times.

Datasets creation

In the following section we have reported the data used for the creation of the three different cases (a small, a medium and a large dataset) as reported by the guidelines of the assigned project. The only values we have modified are the following ones:

- number of customers;
- number of terminals;
- number of days;
- starting date.

Dataset 1 - small size (100 MB)

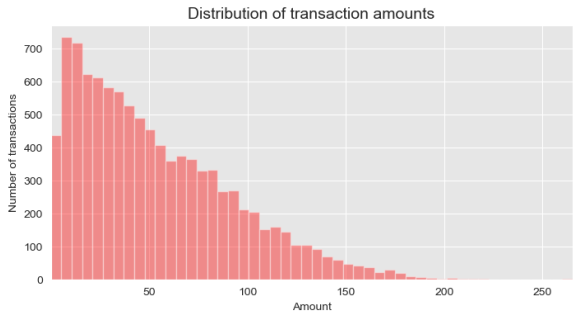
Data:

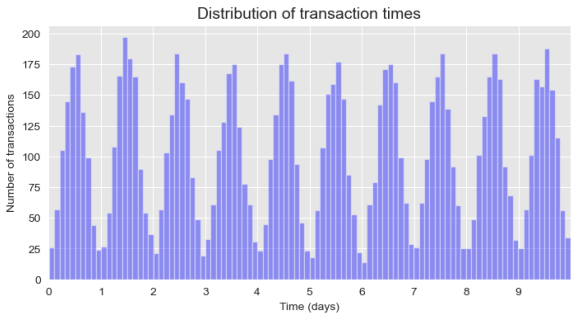
- number of customers: 2500
- number of terminals: 5000
- number of days: 365
- starting date: 2022-10-03
- r: 5

Time to:

- generate customer profiles table: 0.018s
- generate terminal profiles table: 0.011s
- associate terminals to customers: 0.66s
- generate transactions: 7.5e+01s

Number of transactions: 1745300

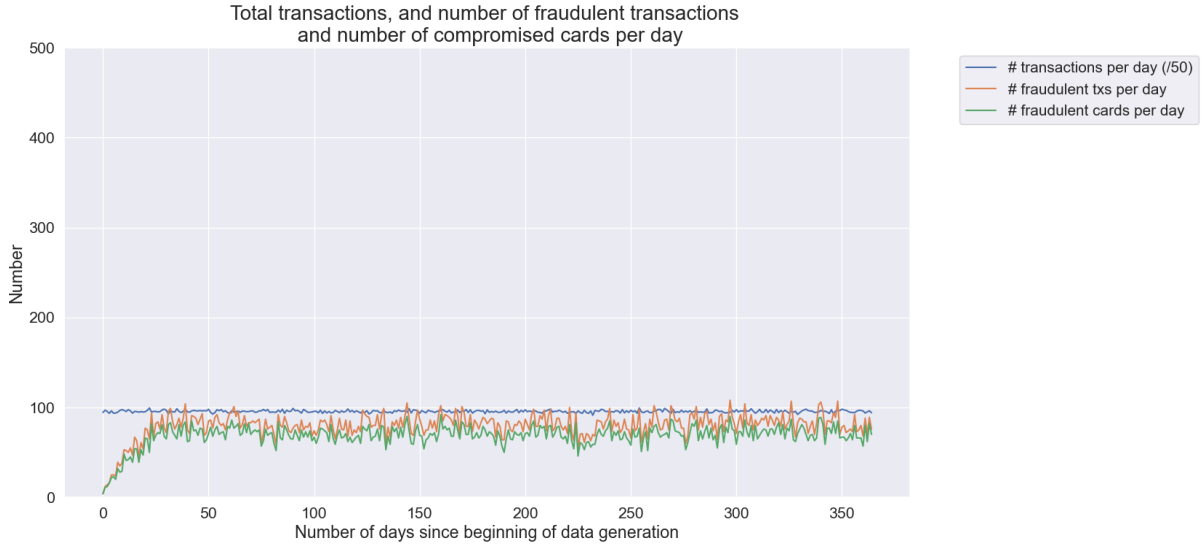




Adding frauds in the dataset (results):

- number of frauds from scenario 1: 892
- number of frauds from scenario 2: 18805
- number of frauds from scenario 3: 9385
- wall time: 2min 6s

Percent of fraudulent transactions: 0.016663037873145017
Number of fraudulent transactions: 29082



Dataset 2 - medium size (205 MB)

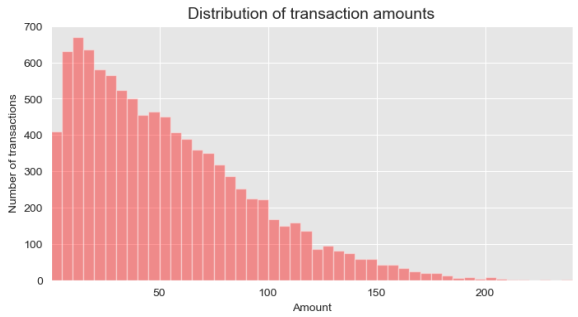
Data:

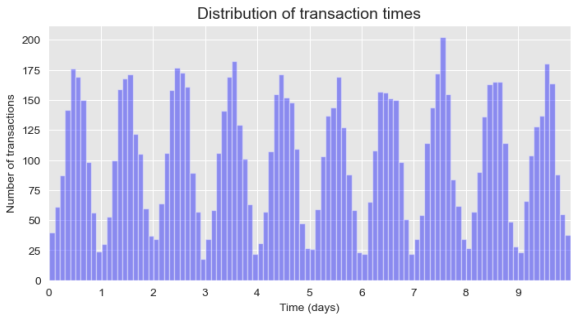
- number of customers: 5000
- number of terminals: 9000
- number of days: 365
- starting date: 2022-10-03
- r: 5

Time to:

- generate customer profiles table: 0.049s
- generate terminal profiles table: 0.049s
- associate terminals to customers: 1.5s
- generate transactions: 1.7e+02s

Number of transactions: 3500393

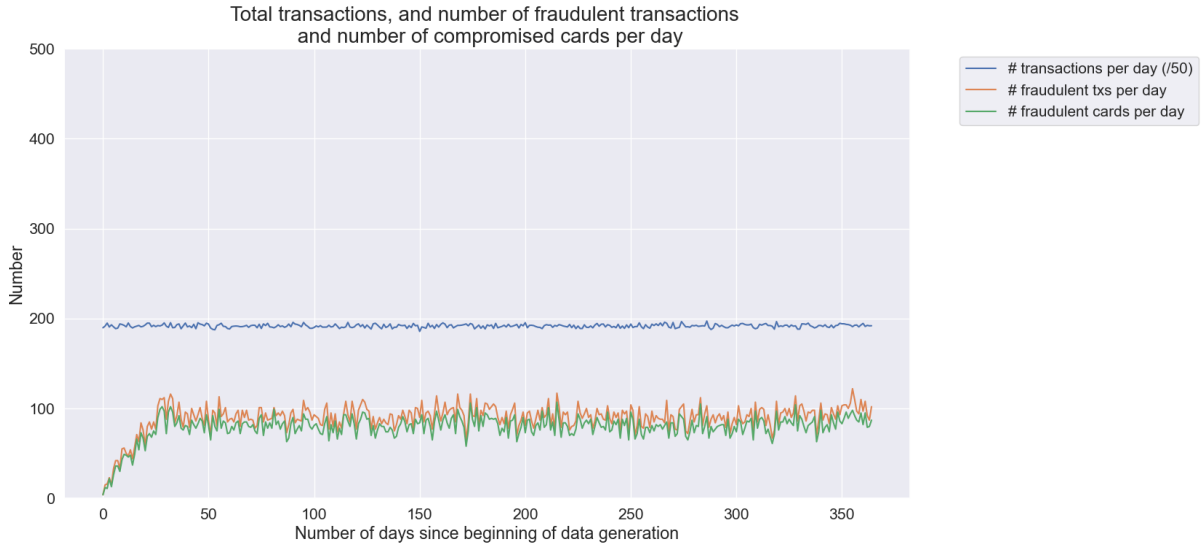




Adding frauds in the dataset (results):

- number of frauds from scenario 1: 1939
- number of frauds from scenario 2: 21335
- number of frauds from scenario 3: 9360
- wall time: 4min 27s

Percent of fraudulent transactions: 0.009322953165544554
Number of fraudulent transactions: 32634



Dataset 3 - large size (314 MB)

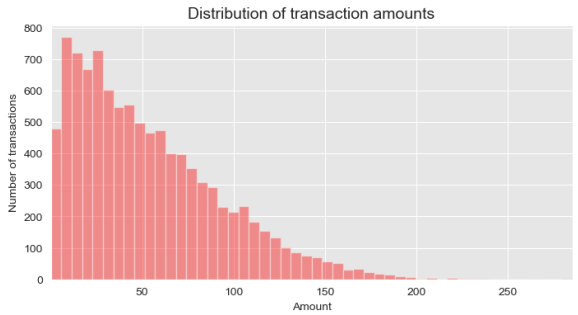
Data:

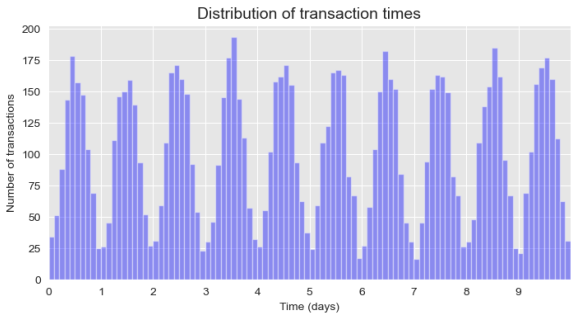
- number of customers: 7500
- number of terminals: 14000
- number of days: 365
- starting date: 2022-10-03
- r: 5

Time to:

- generate customer profiles table: 0.057s
- generate terminal profiles table: 0.057s
- associate terminals to customers: 2.2s
- generate transactions: 2.3e+02s

Number of transactions: 5286887

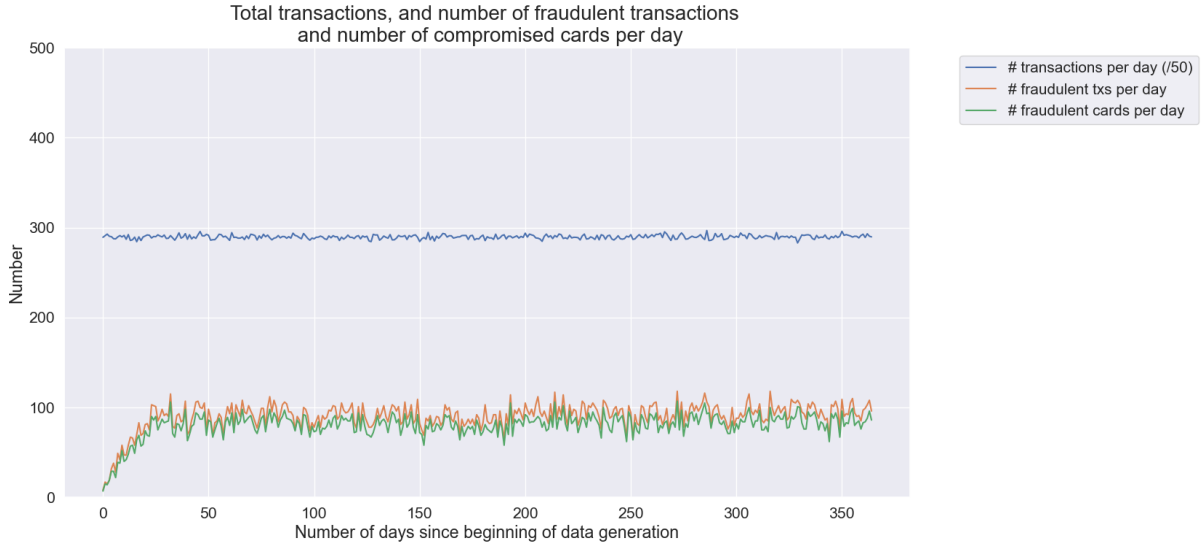




Adding frauds in the dataset (results):

- number of frauds from scenario 1: 2886
- number of frauds from scenario 2: 20436
- number of frauds from scenario 3: 9562
- wall time: 8min 10s

Percent of fraudulent transactions:0.006219917316182472
Number of fraudulent transactions: 32884



Dataset creation time

Time to generate Tables

DATASET	NUM CUSTOMERS	NUM TERMINALS	NUM TRANSACTIONS *generated*	TG CUSTOMER TABLE	TG TERMINAL TABLE	TA TERMINALS TO CUSTOMERS	TG TRANSACTIONS
small	2500	5000	1745300	0,018	0,011	0,66	7,50E+01
medium	5000	9000	3500393	0,049	0,049	1,5	1,70E+02
large	7500	14000	5286887	0,057	0,057	2,2	2,30E+02

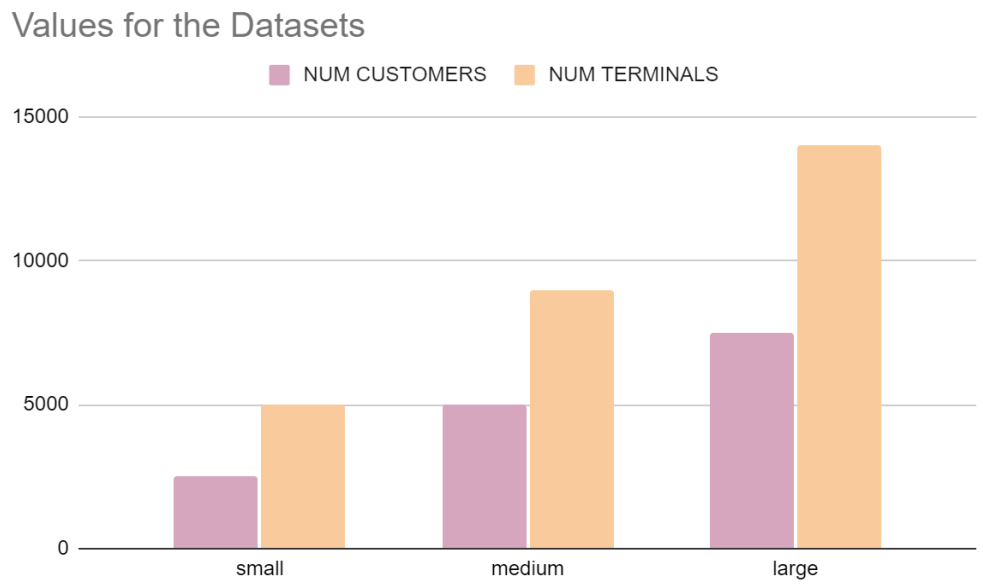
Note:

- *TG = time to generate*
- *TA = time to associate*

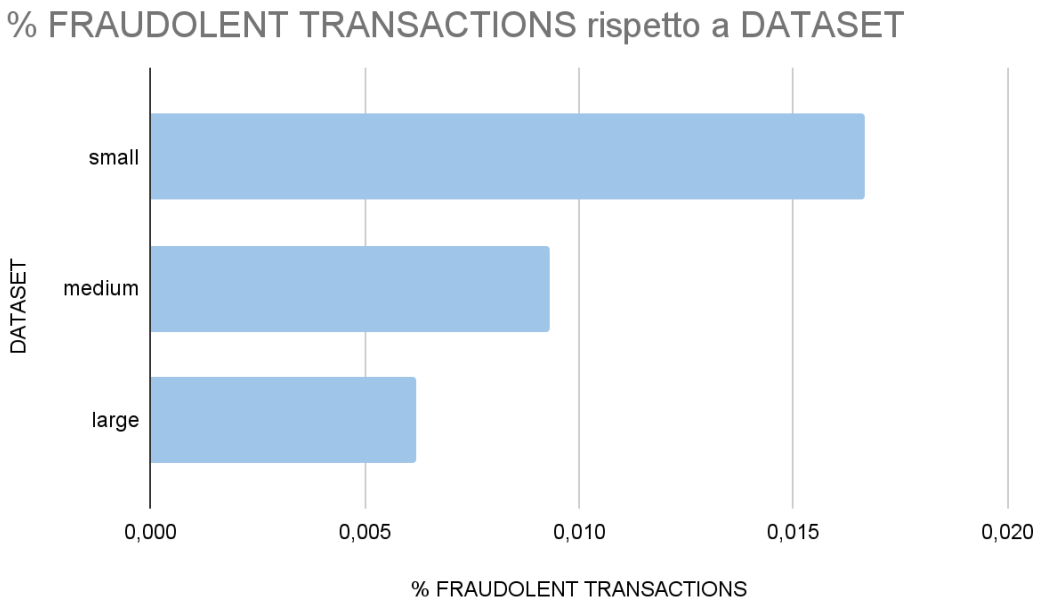
Time is reported in seconds

Data that are equal for all the datasets:

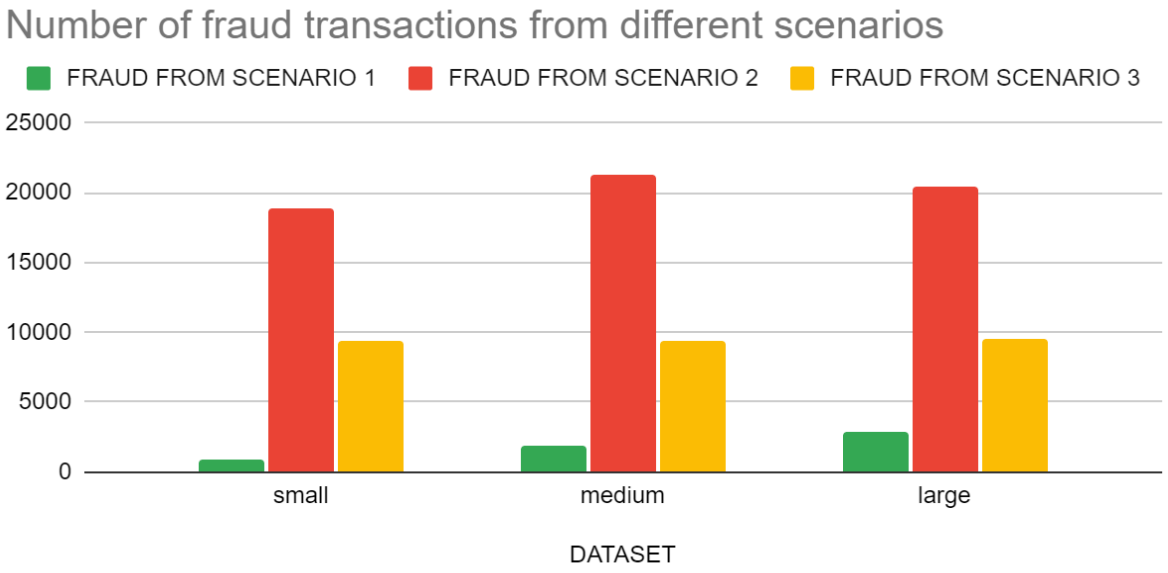
- *number of days = 365*
- *starting date = 2022-10-02*
- *r = 5*



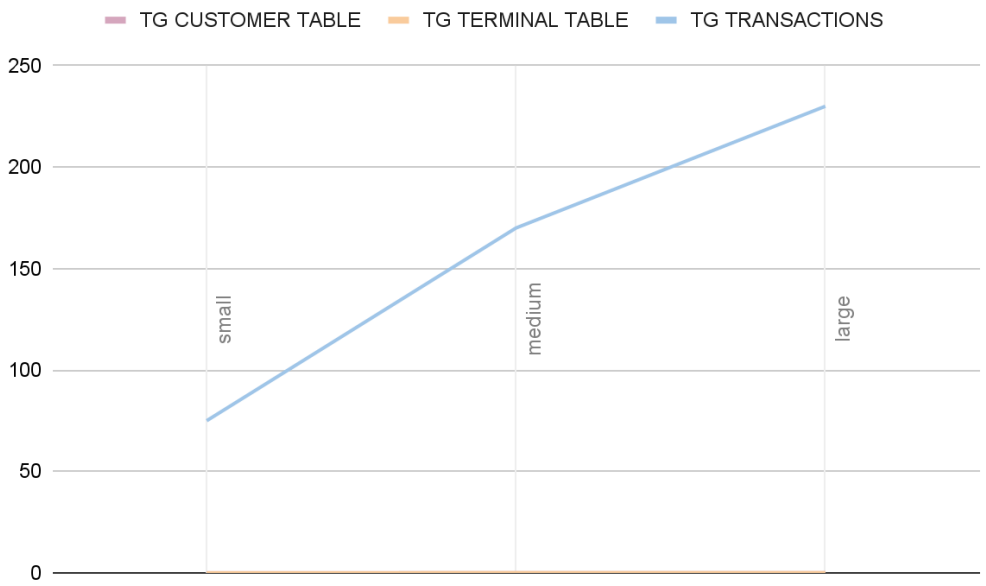
Graph showing the various values used to generate the datasets



Graph in which we report the percentages of fraudulent transactions compared to the total number of transactions that were generated



Graph showing the total of fraudulent transactions, for the various datasets

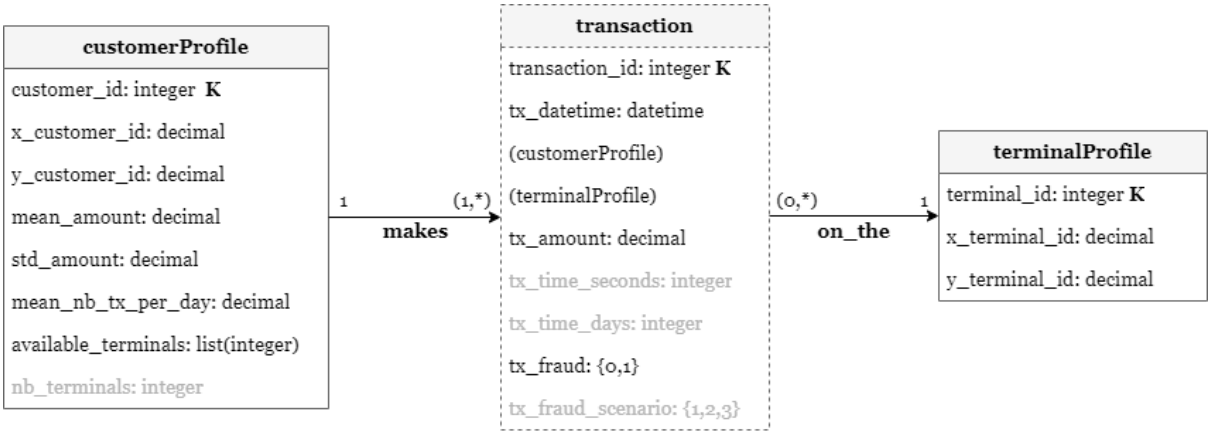


This graph, not very explanatory, has been reported only to point out how much the times relating to the creation of the transitions are very high compared to the creation of the tables for the customers and for the terminals.

UML Class Diagram

Introduction

In the following diagram we have decided to insert not only the fields specified by the project documentation, but also the data used to create the dataset (the ones identified by the grey colour). This choice is due to have a more complete and clear view of the scenario.



Constraints

In the table *customerProfile* there are the following attributes:

- *customer_id* is a unique identifier for each customer;
- (*x_customer_id*, *y_customer_id*) is a pair of real and positive coordinates in a 100·100 grid that defines the geographical location of the customer;
- *mean_amount* and *std_amount* are the mean and standard deviation of the transaction amounts for the customer, assuming that the transaction amounts follow a normal distribution;
    - *mean_amount* is drawn from a uniform distribution (5, 100);
    - *std_amount* is set as the *mean_amount* divided by 2;
- *mean_nb_tx_per_day* is the average number of transactions per day for the customer, assuming that the number of transactions per day follows a Poisson distribution;
- *mean_nb_tx_per_day* is drawn from a uniform distribution (0, 4);
- *available_terminals* is the set of terminals (or more precisely the list of terminal_id defined in the table *terminalProfile*) that the customer can use for making transactions because we assume that the customer only makes transactions on terminals that are geographically close to his geographical location;
    - the value of *available_terminals* is defined by the function *get_list_terminals_within_radius* that takes as input the customer profile (or more precisely the tuple in the table *customerProfile*), an array containing the geographical location of each terminal and the radius r;
- *nb_terminals* is the number of the available terminals for the customer so it is equal to the length of *available_terminals*.

In the table *terminalProfile* there are the following attributes:

- *terminal_id* is a unique identifier for each terminal;
- (*x_terminal_id*, *y_terminal_id*) is a pair of real and positive coordinates in a 100·100 grid that defines the geographical location of the terminal;

In the table *transaction* there are the following attributes:

- a transaction is a payment card transaction consists of any amount paid by a customer to a merchant at a certain time;
- *transaction_id* is a unique identifier for each transaction;
- *tx_datetime* is the date and the time at which the transaction occurs;
- *customer_id* is the unique identifier for the customer (defined in the table *customerProfile*) that paid to a merchant;
- *terminal_id* is the unique identifier for the merchant, or more precisely the terminal (defined in the table *terminalProfile*) to whom the customer paid;
- *tx_amount* is a positive value for the amount of the transaction;
- *tx_time_seconds* is a positive value representing the number of seconds passed from the simulator's starting date (*2022-10-01*) up to the moment in which the transaction takes place;
- *tx_time_days* is a positive value representing the number of days passed from the simulator's starting date (*2022-10-01*) up to the moment in which the transaction takes place;
- *tx_fraud* is a binary variable, with the value 0 for a legitimate transaction or the value 1 for a fraudulent transaction;
- *tx_fraud_scenario* is a positive value in the range [1, 3] representing one of the three possible fraud scenarios used for adding fraudulent transactions to the dataset;
  - in scenario 1, any transaction whose amount is more than 220 is a fraud;
  - in scenario 2, every day, a list of two terminals is drawn at random and all the transactions on these terminals in the next 28 days will be marked as fraudulent;
  - in scenario 3, every day, a list of 3 customers is drawn at random and in the next 14 days, ⅓ of their transactions have their amounts multiplied by 5 and marked as fraudulent.

A customer can pay a merchant by using his terminal only if that terminal is included in the customer's list *available_terminals*.

Since a customer can execute many transactions, we have decided to create an entity instead of a relationship.

A transaction is associated with only one customer.

A transaction can be executed only on one terminal.

In a realistic scenario these are the possible outcomes:

- a customer does not have any terminals near and for this reason he can not make any transaction;
- a terminal is not used for a specific period of time.

For these reasons the minimum cardinality of the relationships (*makes* and *on_the*) are initialised at 0. However the script used by the simulator uses random functions, that is why we do not have the certainty that this happens.

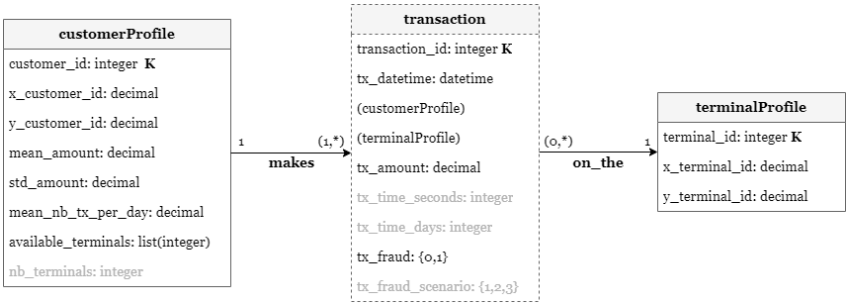
A customer can execute at least one transaction because of the script used by the simulator for generating the dataset.

Logical Data Model

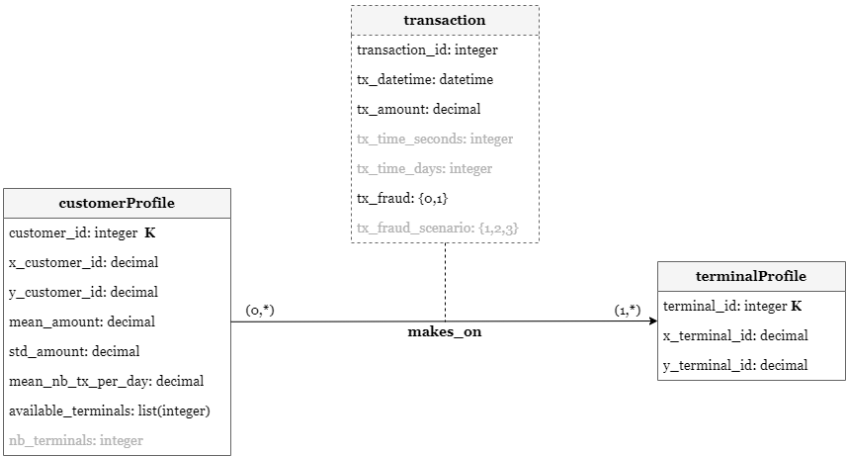
Implementation choices

As explained in the previous section, since a customer can make many transactions, we have represented each transaction as an entity (Data Model 1) and not as a relationship between customer and terminal (Data Model 2).

Data Model 1 (used for the project):



Data Model 2:



The decision to adopt the Data Model 1 led to the creation of two different relationships, one between customer and transaction (*makes*) and another one between transaction and terminal (*on_the*) rather than a single relationship between customer and terminal (*makes_on*) like the Data Model 2.

Although the number of relationships in the Data Model 1 is twice the number of relationships in the Data Model 2, this implementation choice has allowed us to gain advantages in the execution of the queries. In fact most of the queries in the provided document do not involve the three entities at the same time. As a result, having two relationships instead of only one, has allowed us to reduce the amount of information to access.

Graph database and Neo4j

We have decided to organise data according to a graph because we have to maintain information about the relationships between the nodes and the queries in the workload needed to navigate through these relationships. In this sense the connection between customers, terminals and transactions can be represented by edges, while the classes themselves take shape through the concept of node. Furthermore financial services, like fraud detection, are common use cases for graph databases.

Neo4j is one of the data management systems developed for working with graph databases. In particular it works on property graphs, graphs with labels and/or properties for both nodes and edges. Neo4j is more suitable for some big data and analytics applications than row and column databases or free-form JSON document databases for many use cases.

Cypher

CSV files loading

In this section we have reported code used for loading the csv files into our DBMS. All the grey attributes in the UML (seen before) are not included in the following lines.

Consideration

In the following lines of code we have used the `LOAD CSV` clause to import data to the graph. `CALL{...}` allows the execution of subqueries (queries inside of other queries). Furthermore subqueries can be made to execute in separate, inner transactions, producing intermediate commits and this is useful with large write operations. In order to execute a subquery in separate transactions, we have added `IN TRANSACTIONS` after the subquery.

Customer

```
:auto
LOAD CSV WITH HEADERS FROM 'file:///customer_profiles_table.csv' AS line
CALL{
    WITH line
    CREATE (:Customer {
        id: toInteger(line.CUSTOMER_ID),
        x: toFloat(line.x_customer_id),
        y: toFloat(line.y_customer_id),
        mean_amount: toFloat(line.mean_amount),
        std_amount: toFloat(line.std_amount),
        mean_nb_tx_per_day: toFloat(line.mean_nb_tx_per_day),
        available_terminals:
split(replace(replace(replace(line.available_terminals, "[", ""), "]", ""), "
", ""),","), nb_terminals:toInteger(line.nb_terminals)
    })
} IN TRANSACTIONS
```

Conversion of *available_terminals* from array of strings to array of integers

```
MATCH (c:Customer)
UNWIND c.available_terminals AS t
WITH c, collect(toInteger(t)) AS list
SET c.available_terminals = list
RETURN c.available_terminals
```

Terminal

```
:auto
LOAD CSV WITH HEADERS FROM 'file:///terminal_profiles_table.csv' AS line
CALL{
    WITH line
    CREATE (:Terminal {id: toInteger(line.TERMINAL_ID),
    x: toFloat(line.x_terminal_id),
    y: toFloat(line.y_terminal_id)})
```

```
} IN TRANSACTIONS
```

Transaction

```
:auto
LOAD CSV WITH HEADERS FROM 'file:///transactions_df.csv' AS line
CALL{
    WITH line
    CREATE (tx:Transaction {
        id: toInteger(line.TRANSACTION_ID),
        datetime: line.TX_DATETIME,
        amount: toFloat(line.TX_AMOUNT),
        fraud: toInteger(line.TX_FRAUD)
    })
    WITH tx, toInteger(line.CUSTOMER_ID) AS cid,
    toInteger(line.TERMINAL_ID) AS tid
    MATCH (c:Customer {id:cid}),(t:Terminal {id:tid})
    MERGE (c)-[:MAKES]->(tx)-[:ON_THE]->(t)
} IN TRANSACTIONS
```

Dataset loading times

During the loading phase we realised that the times were very high, so looking at the documentation we decided to modify the parameters of the databases.
We report the updated values below.

For the **small** dataset we used:

- dbms.memory.heap.initial_size=**4G**
- dbms.memory.heap.max_size=**4G**

For the **medium** dataset we used:

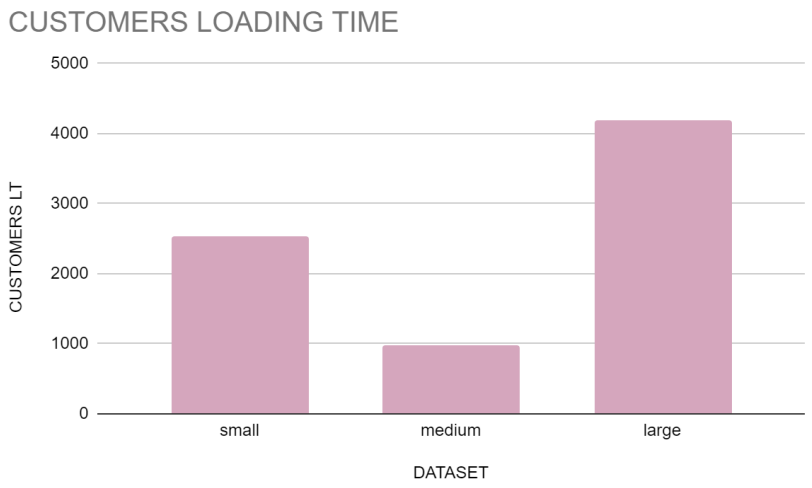
- dbms.memory.heap.initial_size=**6G**
- dbms.memory.heap.max_size=**6G**

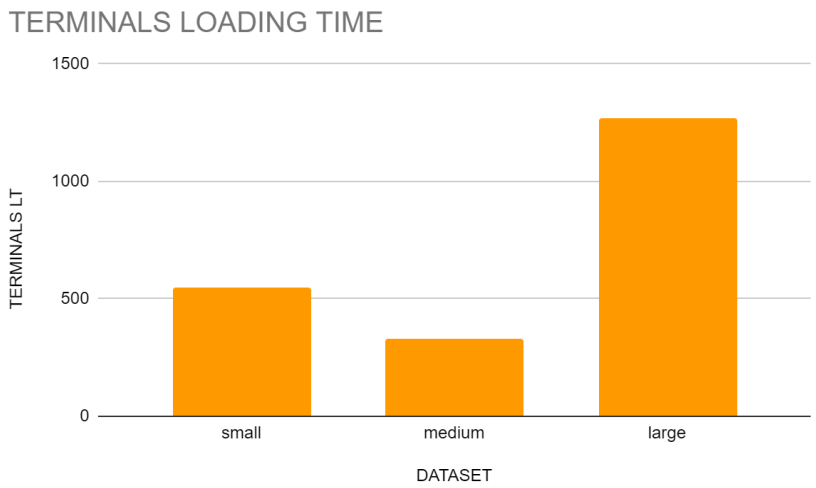
For the **large** dataset we used:

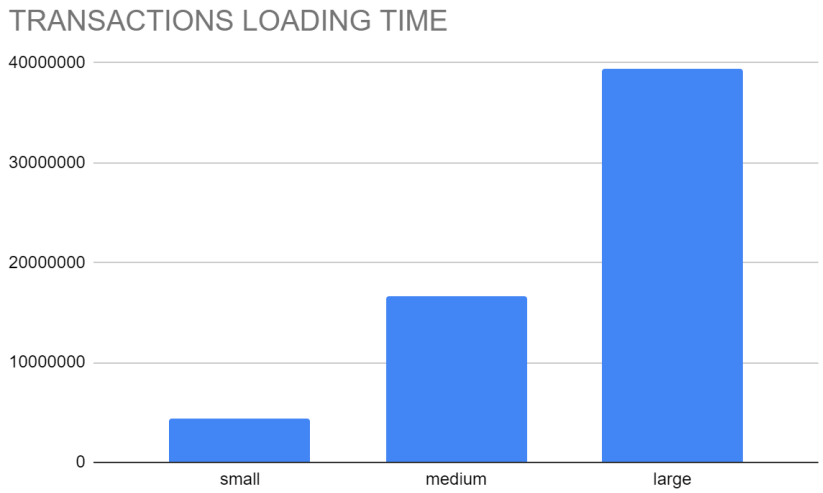
- dbms.memory.heap.initial_size=**8G**
- dbms.memory.heap.max_size=**8G**

Note:
LT = Loading Time
The Loading Time is reported in milliseconds
The time reported for the TRANSACTIONS LT includes also the creation of the two relationships

DATASET	CUSTOMERS LT	TERMINALS LT	TRANSACTIONS LT
small	2524	550	4454819
medium	980	330	16645816
large	4183	1270	39429964


CUSTOMERS LOADING TIME
5000
4000
3000
2000
1000
0
CUSTOMERS LT
small
medium
large
DATASET


TERMINALS LOADING TIME
1500
1000
500
0
TERMINALS LT
small
medium
large
DATASET


TRANSACTIONS LOADING TIME
40000000
30000000
20000000
10000000
0
small
medium
large

Queries

General optimization

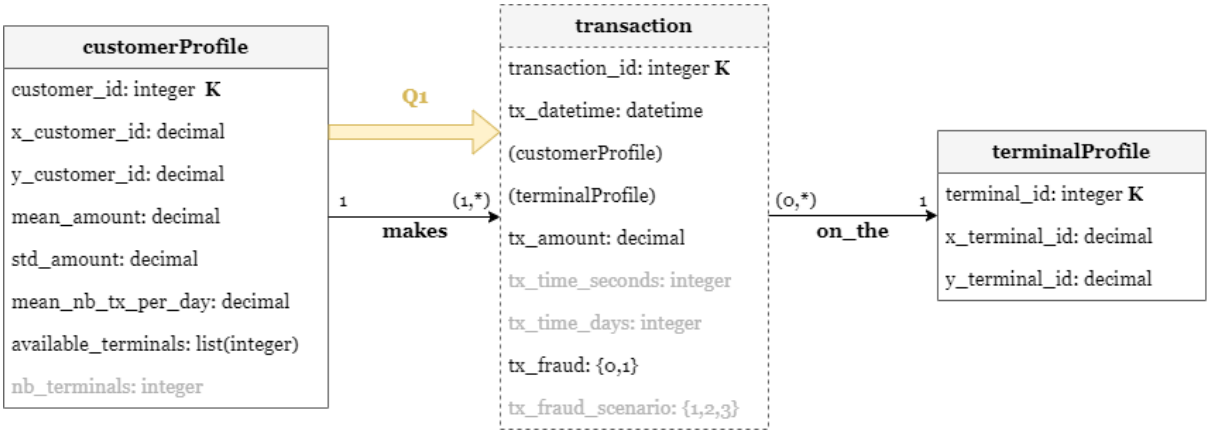
Given that for all queries we are going to be looking up IDs of transactions, customers and terminals, we have decided to optimise the execution by creating indexes for reaching better performances.

```
CREATE INDEX IF NOT EXISTS FOR (tx:Transaction) ON (tx.id)

CREATE INDEX IF NOT EXISTS FOR (c:Customer) ON (c.id)

CREATE INDEX IF NOT EXISTS FOR (t:Terminal) ON (t.id)
```

Query 1

For each customer, identify the amount that he/she has spent for every week of the current semester.

UML Class Diagram



Considerations

In order to create a flexible query, we have decided to use the function `date()` that returns the current date value. Starting from this date, we have identified the current semester and the relative year.

Furthermore, since our dataset starts from October 2022, we have also checked that if the current semester is the second one, the months to take into account are only the ones in the same year of the current date and not the ones in 2022.

Query optimization

In this query we are going to be looking up transactions' datetime frequently. Therefore, for better performance, we have created an index on *datetime* property for the *Transaction* node in the following way.

```
CREATE INDEX IF NOT EXISTS FOR (tx:Transaction) ON (tx.datetime)
```

Query

```
MATCH (c:Customer)-[:MAKES]->(tx:Transaction)
WITH apoc.date.fields(tx.datetime, 'yyyy-MM-dd HH:mm:ss') AS d, c, tx
WITH date({year: d.years, month: d.months, day: d.days}) AS dtx, c, tx,
date() AS cd
WITH dtx, c, tx, cd,
CASE
  WHEN cd.month<=6 AND dtx.month<=6 AND cd.year=dtx.year THEN tx.amount
  WHEN cd.month>6 AND dtx.month>6 AND cd.year=dtx.year THEN tx.amount
  ELSE NULL
END AS amount
WHERE amount IS NOT NULL
RETURN c.id AS customer_id, dtx.week AS week, sum(amount) AS tot_amount
```

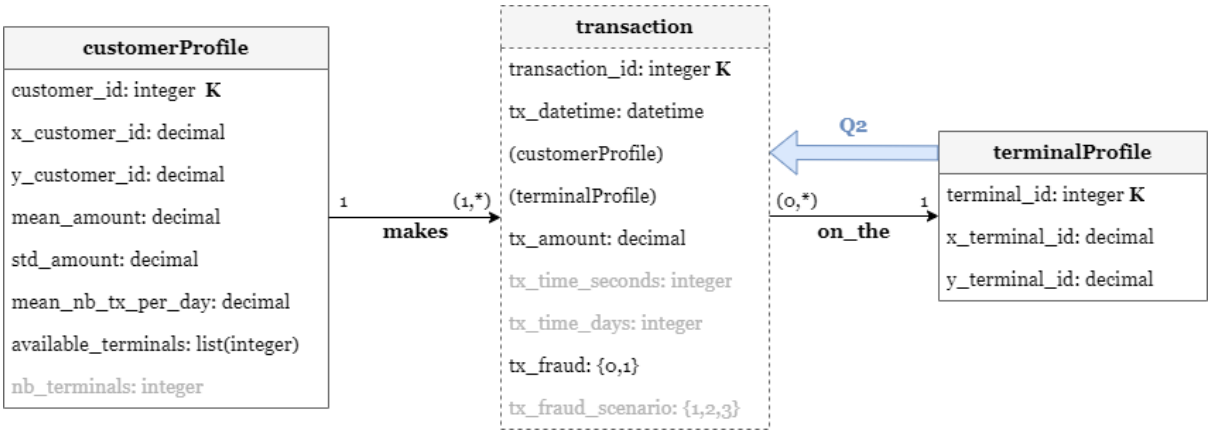
Execution time

Dataset 1 (small)	147 ms
Dataset 2 (medium)	800 ms
Dataset 3 (large)	900 ms

Query 2

For each terminal identify the possible fraudulent transactions. The fraudulent transactions are those whose import is higher or lower than 10% of the average import of the transactions executed on the same terminal in the previous semester.

UML Class Diagram



Considerations

In order to simplify the management of the semesters, we have decided to associate an integer to each semester in the period between October 2022 and October with the following criteria:

- semester 1 = October, November and December 2022
- semester 2 = January, February, March, April, May and June 2023
- semester 3 = July, August, September and October 2023

Since semester 1 is the first semester in our dataset, all transactions carried out in that period have not been considered in the evaluation of fraudulent transactions because the previous semester does not exist. This consideration is applied by the condition `WHERE semester IS NOT NULL` (line 28).

Looking at the data in our datasets, we have noticed that on some terminals no transitions have been executed in some semesters. Since the average of the transactions is computed for each semester with at least one transaction, these values cannot be stored temporarily into an array because the order would not be respected if no transitions have been executed in at least one semester (above all if the missing average is relative to semesters 1 and/or 2). For example, let's consider a terminal where no transactions have been carried out in semester 2. The array associated with this terminal is *[avg1, avg3]* and this would be a problem since the index of each value does not match the id of the semester. Therefore, in order to somehow preserve the order, we have decided to keep also an array with the id of the semesters for which it has been possible to compute the average. These two arrays have been stored temporarily into a map (*map_semesters*) composed of two key-value pairs: *{ semester: [...], avg: [...]}*.

Moreover, still considering the previous example, due to the absence of *avg2* it is impossible to identify which transactions executed in semester 3 are fraudulent or not. Therefore we

have decided not to evaluate the fraudulence of all those transactions carried out in semester i such that in semester i-1 no transaction has been executed. This consideration is applied by the condition `WHERE idx >= 0` (line 32).

Query optimization

In this query we are going to be looking up transactions' datetime frequently. Therefore, for better performance, we have created an index on *datetime* property for the *Transaction* node in the following way (already executed for Q1).

```
CREATE INDEX IF NOT EXISTS FOR (tx:Transaction) ON (tx.datetime)
```

Query

```
MATCH (tx:Transaction)-[r:ON_THE]->(t:Terminal)
CALL{
    WITH tx, r, t
    WITH apoc.date.fields(tx.datetime, 'yyyy-MM-dd HH:mm:ss') AS d, tx, t
    WITH date({year: d.years, month: d.months, day: d.days}) AS dtx, tx, t
    WITH dtx, tx, t,
    CASE
        WHEN dtx.month>=7 AND dtx.year=2022 THEN 1
        WHEN dtx.month>=1 AND dtx.month<7 AND dtx.year=2023 THEN 2
        WHEN dtx.month>=7 AND dtx.year=2023 THEN 3
        ELSE NULL
    END AS semester
    ORDER BY semester

    MATCH (tx)-[r]->(t)
    WITH t, semester, avg(tx.amount) AS avg
    WITH t, {semester:collect(semester), avg:collect(avg)} AS map_semesters

    MATCH (tx)-[r]->(t)
    WITH apoc.date.fields(tx.datetime, 'yyyy-MM-dd HH:mm:ss') AS d, tx, t,
map_semesters
    WITH date({year: d.years, month: d.months, day: d.days}) AS dtx, tx, t,
map_semesters
    WITH dtx, tx, t, map_semesters,
    CASE
        WHEN dtx.month<7 AND dtx.year=2023 THEN 2
        WHEN dtx.month>=7 AND dtx.year=2023 THEN 3
        ELSE NULL
    END AS semester
    WHERE semester IS NOT NULL
    WITH t, tx, semester, semester-1 AS prev_semester, map_semesters
    WITH t, tx, semester, semester-1 AS prev_semester, map_semesters,
apoc.map.get(map_semesters, 'semester') AS k_from_map,
apoc.map.get(map_semesters, 'avg') AS v_from_map
    WITH t, tx, semester, prev_semester, map_semesters, k_from_map,
apoc.coll.indexOf(k_from_map, prev_semester) AS idx, v_from_map
    WHERE idx >= 0
    WITH t, tx, semester, prev_semester, map_semesters, k_from_map, idx,
v_from_map[idx] AS prev_avg
```

```
    WITH t, tx, semester, prev_semester, map_semesters, k_from_map, idx,
prev_avg,
    CASE
        WHEN tx.amount>=prev_avg+prev_avg*0.1 THEN 1
        WHEN tx.amount<=prev_avg-prev_avg*0.1 THEN 1
        ELSE 0
    END AS fraud
    WHERE fraud=1
    RETURN t.id AS id_terminal, collect(tx.id) AS fraud_transactions,
prev_avg ORDER BY t.id
}
RETURN id_terminal, fraud_transactions, prev_avg
```

Execution time

Dataset 1 (small)	1682 ms
Dataset 2 (medium)	3679 ms
Dataset 3 (large)	10743 ms

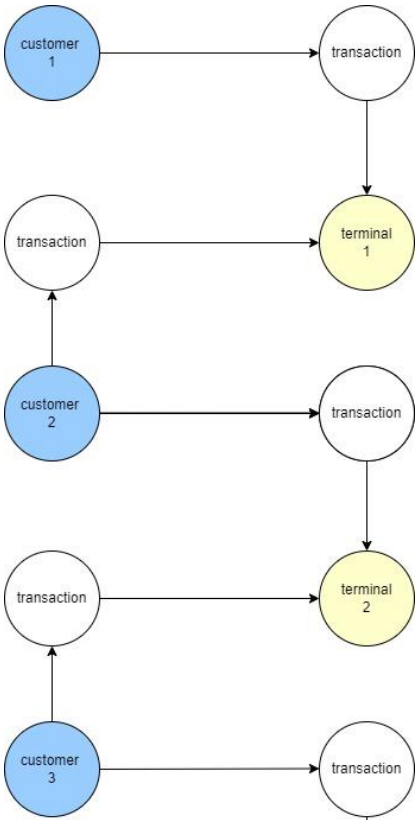
Query 3

Given a user u, determine the "co-customer-relationships CC of degree k". A user u' is a co-customer of u if you can determine a chain "u_1-t_1-u_2-t_2-...t_{k-1}-u_k" such that u_1=u, u_k=u', and for each 1<=i, j<=k, u_i <> u_j, and t_1,..t_{k-1} are the terminals on which a transaction has been executed. Therefore, $CC_k(u)$={u'| a chain exists between u and u' of degree k}. Please, note that depending on the adopted model, the computation of $CC_k(u)$ could be quite complicated. Consider at least the computation of $CC_3(u)$ (i.e. the co-customer relationships of degree 3).

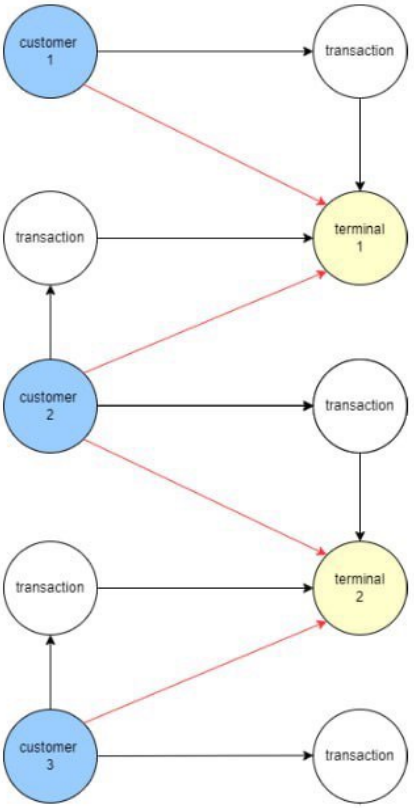
Considerations

Our scenario is represented by the following image: each customer *MAKES* a transaction *ON* a specific terminal. In order to simplify the query, we have created a new relationship called *MAKES_ON* that connects directly a customer to a terminal.

Before the creation of *MAKES_ON*



After the creation of *MAKES_ON*



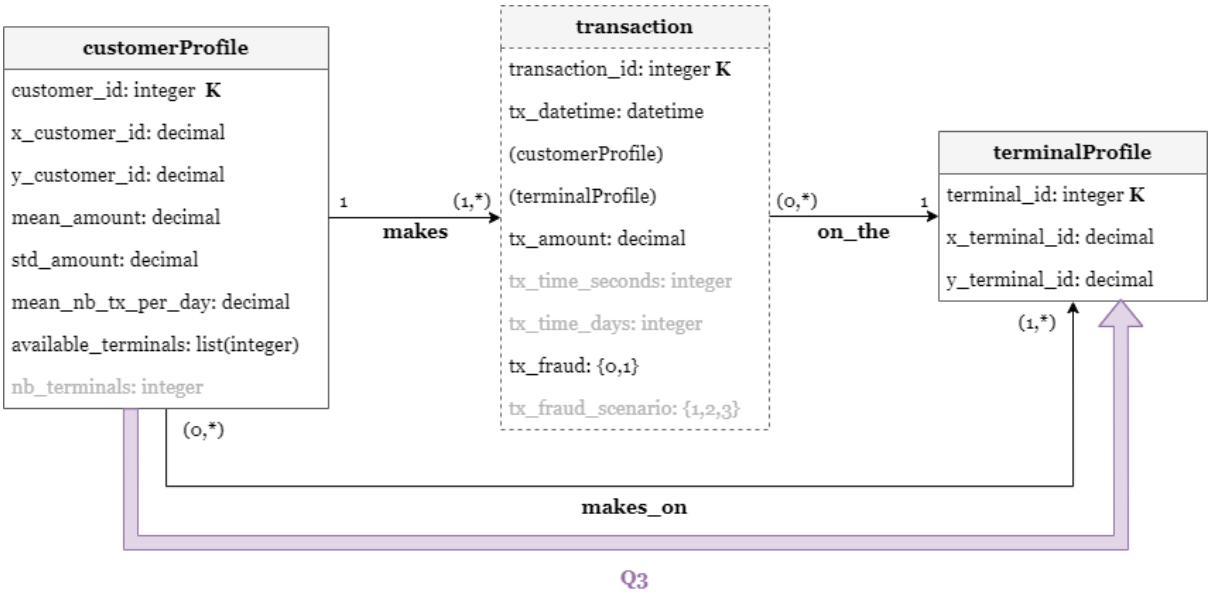
Relationship creation

```
:auto
MATCH (c:Customer)-[:MAKES]->(:Transaction)-[:ON_THE]->(t:Terminal)
CALL {
    WITH c, t
    MERGE (c)-[r:MAKES_ON]->(t)
} IN TRANSACTIONS
```

Execution time (relationship creation)

Dataset 1 (small)	16318 ms
Dataset 2 (medium)	91740 ns
Dataset 3 (large)	181697 ms

UML Class Diagram



Considerations

Suppose that nodes c_1 and c_2 are co-customer of degree k=3. The result of the query should not report both the pairs (c_1, c_2) and (c_2, c_1). Therefore we have added the condition `WHERE c1.id<c2.id` in order to obtain each pair only once. The idea is that for each node we have to look only at the nodes with an ID greater than its one because all the possible co-customers with an ID lower than its one have been already returned.

Query

```
MATCH (c1:Customer)-[r:MAKES_ON*3]-(c2:Customer)
WHERE c1.id<c2.id
RETURN DISTINCT c1.id, c2.id
```

Execution time (query)

Dataset 1 (small)	12594 ms*
Dataset 2 (medium)	44400 ms*
Dataset 3 (large)	154885 ms

*Observation

These execution times refer to the query performed with k=4. In fact, unlike the large dataset, the execution of the query on the small and on the medium ones did not produce any result for k=3. Therefore we have deduced that there is no co-customer-relationship of degree 3 in these two datasets.

Query 4

Extend the logical model that you have stored in the NOSQL database by introducing the following information (pay attention that this operation should be done once the NOSQL database has been already loaded with the data extracted from the datasets).

Query 4.1 A

Each transaction should be extended with the period of the day {morning, afternoon, evening, night} in which the transaction has been executed.

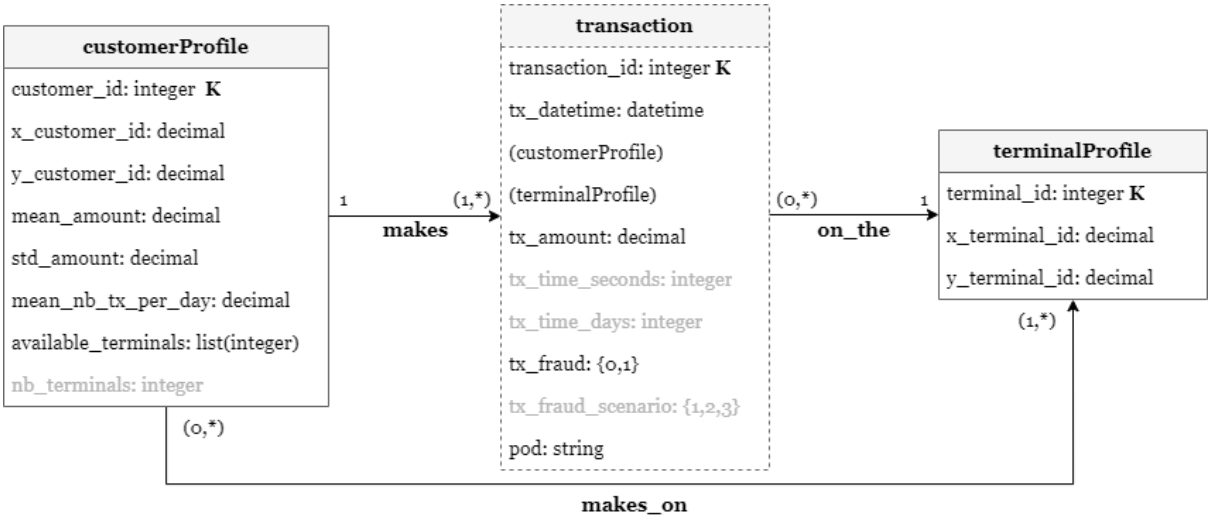
Considerations

The different Period Of the Day (POD) are so divided:

- morning = [5.00, 12.00)
- afternoon = [12.00, 17.00)
- evening = [17.00, 21.00)
- night = [21.00, 5.00)

UML Class Diagram

The updated UML is the following one:



Query optimization

In this query we are going to be looking up transactions' datetime frequently. Therefore, for better performance, we have created an index on *datetime* property for the *Transaction* node in the following way (already executed for Q1).

```
CREATE INDEX IF NOT EXISTS FOR (tx:Transaction) ON (tx.datetime)
```

Query

```
:auto
MATCH (tx:Transaction)
CALL{
    WITH tx
```

```
    WITH tx,apoc.date.fields(tx.datetime,'yyyy-MM-dd HH:mm:ss') AS d
    WITH tx, d.hours AS h
    WITH tx, h,
    CASE
        WHEN h>=5 AND h<12 THEN 'morning'
        WHEN h>=12 AND h<17 THEN 'afternoon'
        WHEN h>=17 AND h<21 THEN 'evening'
        WHEN h>=21 OR h<5 THEN 'night'
        ELSE NULL
    END AS pod
    SET tx.pod=pod
} IN TRANSACTIONS
```

Execution time

Dataset 1 (small)	16529 ms
Dataset 2 (medium)	23976 ms
Dataset 3 (large)	157098 ms

Query 4.1 B

Each transaction should be extended with the kind of products that have been bought through the transaction {high tech, food, clothing, consumable, other}. The values can be chosen randomly.

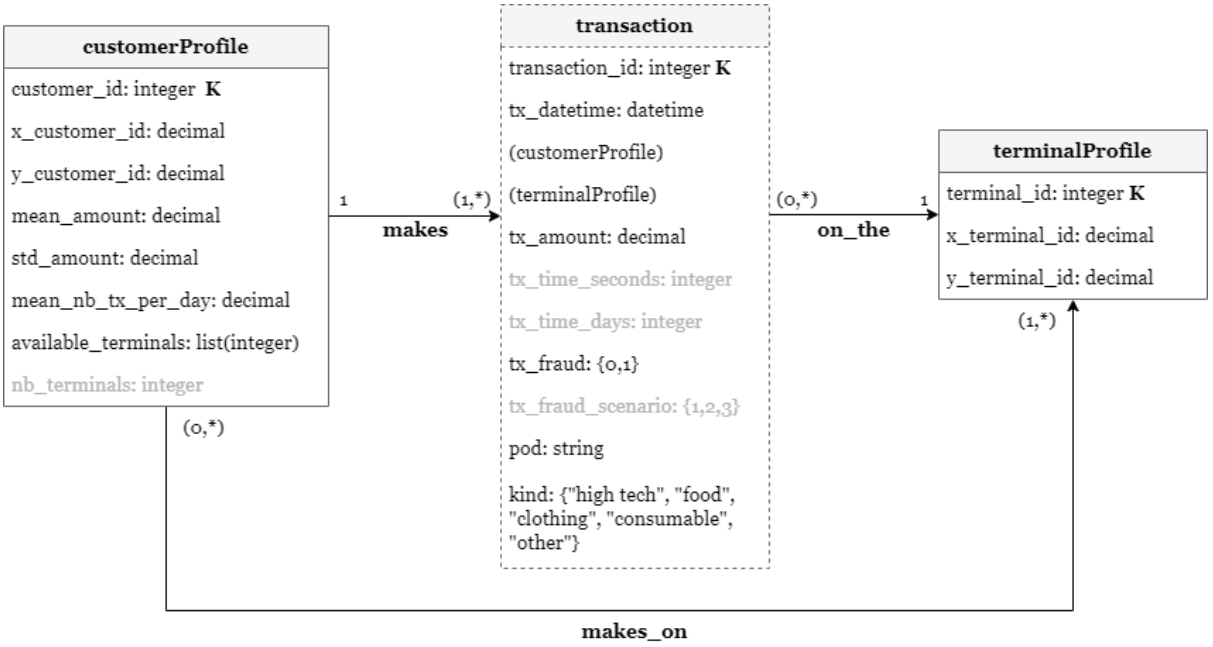
Considerations

The function `apoc.text.random()` generates a random string taking as input a length parameter and an optional string of valid characters. We have provided a string of numbers with the following correlations:

- 1 = high tech
- 2 = food
- 3 = clothing
- 4 = consumable
- 5 = other

UML Class Diagram

The updated UML is the following one:



Query

```
:auto
MATCH (tx:Transaction)
CALL{
    WITH tx
    WITH tx, apoc.text.random(1, '12345') AS r
    WITH tx, r,
    CASE
        WHEN r='1' THEN 'high tech'
        WHEN r='2' THEN 'food'
        WHEN r='3' THEN 'clothing'
        WHEN r='4' THEN 'consumable'
```

```
        WHEN r='5' THEN 'other'
    ELSE NULL
    END AS kind
    SET tx.kind=kind
} IN TRANSACTIONS
```

Execution time

Dataset 1 (small)	12629 ms
Dataset 2 (medium)	41299 ms
Dataset 3 (large)	124398 ms

Query 4.2

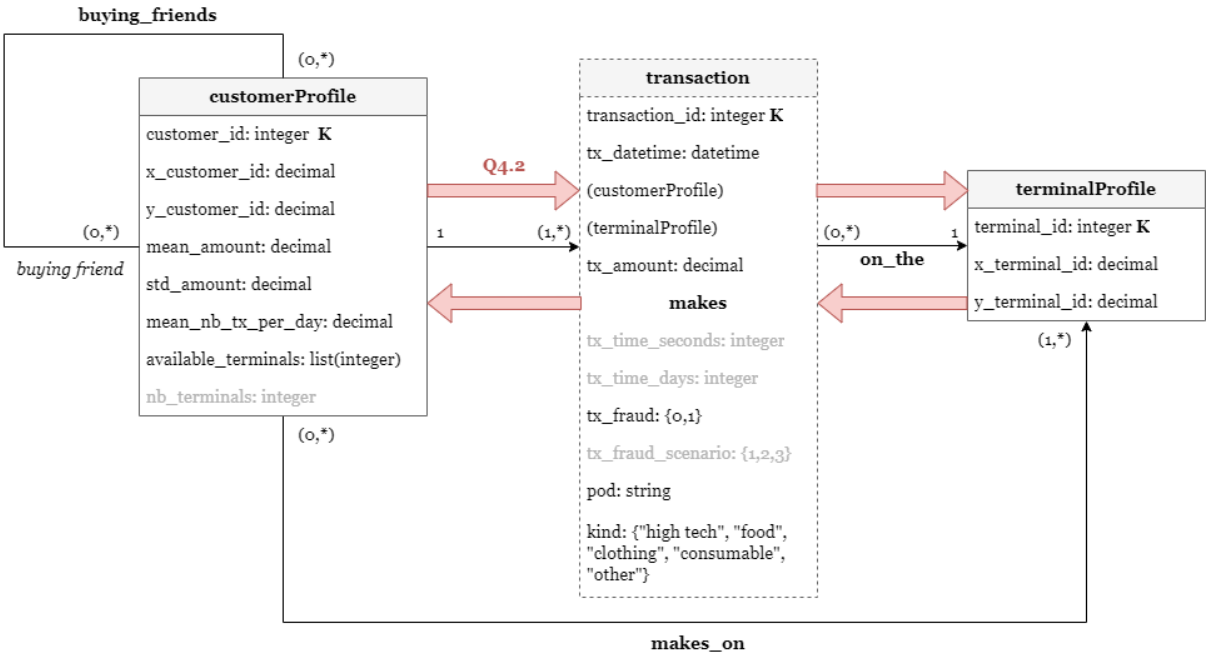
Customers that make more than three transactions related to the same types of products from the same terminal should be connected as "buying_friends". Therefore also this kind of relationship should be explicitly stored in the NOSQL database and can be queried.

Considerations

The new relationship *BUYING_FRIENDS* is not oriented. Furthermore we have removed all the loops by considering only couples of customers different from each other. This consideration is applied by the condition `a.id<>b.id`.

UML Class Diagram

The updated UML is the following one:



Query optimization

In this query we are going to be looking up transactions' kind frequently. Therefore, for better performance, we have created an index on *kind* property for the *Transaction* node in the following way.

```
CREATE INDEX IF NOT EXISTS FOR (tx:Transaction) ON (tx.kind)
```

Query

```
MATCH (a:Customer)-[:MAKES]->(txa:Transaction)-[:ON_THE]->(t:Terminal)
WITH t, a, txa.kind AS kind, count(txa.id) AS nr_txa

MATCH (b:Customer)-[:MAKES]->(txb:Transaction)-[:ON_THE]->(t)
WHERE txb.kind=kind AND a.id<>b.id
WITH t, b, kind, count(txb.id) AS nr_txb, a, nr_txa

WHERE nr_txa>3 AND nr_txb>3
```

```
MERGE (a)-[:BUYING_FRIENDS]-(b)
```

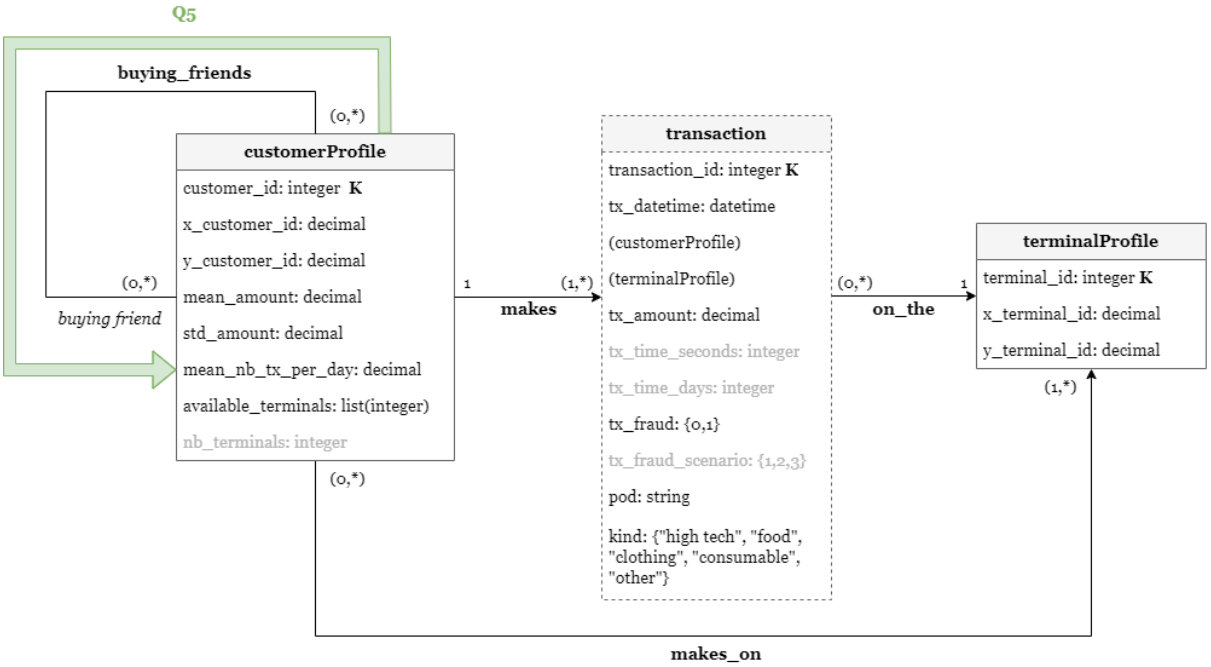
Execution time

Dataset 1 (small)	278817 ms
Dataset 2 (medium)	307298 ms
Dataset 3 (large)	642345 ms

Query 5

Identify the buying-friend of degree K (i.e. the customers that are related through a chain of buying-friend relationships of degree K). Again, depending on the model the operations can become quite complicated. We expect that you write a script for K=4.

UML Class Diagram



Considerations

Suppose that nodes c_1 and c_2 are buying-friends of degree K=4. The result of the query should not report both the pairs (c_1, c_2) and (c_2, c_1). Therefore we have added the condition `WHERE c1.id<c2.id` in order to obtain each pair only once. The idea is that for each node we have to look only at the nodes with an ID greater than its one because all the possible buying-friends with an ID lower than its one have been already returned.

Query

```
MATCH (c1:Customer)-[:BUYING_FRIENDS*4]-(c2:Customer)
WHERE c1.id<c2.id
RETURN DISTINCT c1.id, c2.id
```

Execution time

Dataset 1 (small)	648 ms
Dataset 2 (medium)	704 ms
Dataset 3 (large)	1856 ms

Queries execution time

The graph below shows all the execution times of the queries provided by the document of the project.